Fast generic MCMC for targets with expensive likelihoods

C.Sherlock (Lancaster), A.Golightly (Ncl) & D.Henderson (Ncl); this talk by: Jere Koskela (Warwick)

Monte Carlo Techniques, Paris, July 2016

Motivation

Metropolis-Hastings (MH) algorithms create a realisation: $\theta^{(1)}, \theta^{(2)}, \ldots$ from a Markov chain with stationary density $\pi(\theta)$. For each $\theta^{(i)}$ we evaluate $\pi(\theta^{(i)})$ - and then discard it.

Pseudo-marginal MH algorithms create a realisation of $\hat{\pi}(\theta^{(i)}, u^{(i)})$ - and then discard it.

In many applications evaluating $\pi(\theta)$ or $\hat{\pi}(\theta, u)$ is computationally expensive.

We would like to reuse these values to create a more efficient MH algorithm that still targets the correct stationary distribution.

Motivation

Metropolis-Hastings (MH) algorithms create a realisation: $\theta^{(1)}, \theta^{(2)}, \ldots$ from a Markov chain with stationary density $\pi(\theta)$. For each $\theta^{(i)}$ we evaluate $\pi(\theta^{(i)})$ - and then discard it.

Pseudo-marginal MH algorithms create a realisation of $\hat{\pi}(\theta^{(i)}, u^{(i)})$ - and then discard it.

In many applications evaluating $\pi(\theta)$ or $\hat{\pi}(\theta, u)$ is computationally expensive.

We would like to reuse these values to create a more efficient MH algorithm that still targets the correct stationary distribution.

We will focus on the [pseudo-marginal] random walk Metropolis (RWM) with a Gaussian proposal:

$$\theta' | \theta \sim \mathrm{N}(\theta, \lambda^2 V).$$

This talk

- Creating an approximation to $\pi(\theta')$: k-NN.
- Using an approximation: Delayed acceptance [PsM]MH.
- Storing the values: KD-trees.
- Algorithm: adaptive da-[PsM]MH; ergodicity.
- Choice of P(fixed kernel).
- Simulation study.

Creating an approximation

At iteration n of the MH we have $\pi(\theta^{(1)}), \pi(\theta^{(2)}), \ldots, \pi(\theta^{(n)})$, and we would like to create $\hat{\pi}_a(\theta') \approx \pi(\theta')$.

Creating an approximation

At iteration n of the MH we have $\pi(\theta^{(1)}), \pi(\theta^{(2)}), \ldots, \pi(\theta^{(n)})$, and we would like to create $\hat{\pi}_a(\theta') \approx \pi(\theta')$.

Gaussian process to $\log \pi$?:

Creating an approximation

At iteration n of the MH we have $\pi(\theta^{(1)}), \pi(\theta^{(2)}), \ldots, \pi(\theta^{(n)})$, and we would like to create $\hat{\pi}_a(\theta') \approx \pi(\theta')$.

Gaussian process to $\log \pi$?:
[problems with cost of fitting and evaluating as $n \uparrow$]

Creating an approximation

At iteration n of the MH we have $\pi(\theta^{(1)}), \pi(\theta^{(2)}), \ldots, \pi(\theta^{(n)})$, and we would like to create $\hat{\pi}_a(\theta') \approx \pi(\theta')$.

Gaussian process to $\log \pi$?:
[problems with cost of fitting and evaluating as $n \uparrow$]

Weighted average of **k-nearest neighbour** π values:
(i) Fitting cost: 0 (actually $\mathcal{O}(n_0)$).

Creating an approximation

At iteration n of the MH we have $\pi(\theta^{(1)}), \pi(\theta^{(2)}), \ldots, \pi(\theta^{(n)})$, and we would like to create $\hat{\pi}_a(\theta') \approx \pi(\theta')$.

Gaussian process to $\log \pi$?:
[problems with cost of fitting and evaluating as $n \uparrow$]

Weighted average of **k-nearest neighbour** π values:
(i) Fitting cost: 0 (actually $\mathcal{O}(n_0)$).
(ii) Per-iteration cost: $\mathcal{O}(n)$.

Creating an approximation

At iteration n of the MH we have $\pi(\theta^{(1)}), \pi(\theta^{(2)}), \ldots, \pi(\theta^{(n)})$, and we would like to create $\hat{\pi}_a(\theta') \approx \pi(\theta')$.

Gaussian process to $\log \pi$?:
[problems with cost of fitting and evaluating as $n \uparrow$]

Weighted average of **k-nearest neighbour** π values:
(i) Fitting cost: 0 (actually $\mathcal{O}(n_0)$).
(ii) Per-iteration cost: $\mathcal{O}(n)$.
(iii) Accuracy $\uparrow$ with n.

Delayed acceptance MH (1)

(Christen and Fox, 2005). Suppose we have a computationally-cheap approximation to the posterior: $\hat{\pi}_a(\theta)$.

Delayed acceptance MH (1)

(Christen and Fox, 2005). Suppose we have a computationally-cheap approximation to the posterior: $\hat{\pi}_a(\theta)$.

Define $\alpha_{da}(\theta;\theta') := \alpha_1(\theta;\theta')\ \alpha_2(\theta;\theta')$, where

$$\alpha_1 := 1 \wedge \frac{\hat{\pi}_a(\theta')q(\theta|\theta')}{\hat{\pi}_a(\theta)q(\theta'|\theta)} \quad \text{and} \quad \alpha_2 := 1 \wedge \frac{\pi(\theta')/\hat{\pi}_a(\theta')}{\pi(\theta)/\hat{\pi}_a(\theta)}.$$

Detailed balance (with respect $\pi(\theta)$) is still preserved with α_{da} because

Delayed acceptance MH (1)

(Christen and Fox, 2005). Suppose we have a computationally-cheap approximation to the posterior: $\hat{\pi}_a(\theta)$.

Define $\alpha_{da}(\theta; \theta') := \alpha_1(\theta; \theta')\ \alpha_2(\theta; \theta')$, where

$$\alpha_1 := 1 \wedge \frac{\hat{\pi}_a(\theta')q(\theta|\theta')}{\hat{\pi}_a(\theta)q(\theta'|\theta)} \quad \text{and} \quad \alpha_2 := 1 \wedge \frac{\pi(\theta')/\hat{\pi}_a(\theta')}{\pi(\theta)/\hat{\pi}_a(\theta)}.$$

Detailed balance (with respect $\pi(\theta)$) is still preserved with α_{da} because

$$\pi(\theta)\ q(\theta'|\theta)\ \alpha_{da}(\theta; \theta')$$

Delayed acceptance MH (1)

(Christen and Fox, 2005). Suppose we have a computationally-cheap approximation to the posterior: $\hat{\pi}_a(\theta)$.

Define $\alpha_{da}(\theta;\theta') := \alpha_1(\theta;\theta')\ \alpha_2(\theta;\theta')$, where

$$\alpha_1 := 1 \wedge \frac{\hat{\pi}_a(\theta')q(\theta|\theta')}{\hat{\pi}_a(\theta)q(\theta'|\theta)} \quad \text{and} \quad \alpha_2 := 1 \wedge \frac{\pi(\theta')/\hat{\pi}_a(\theta')}{\pi(\theta)/\hat{\pi}_a(\theta)}.$$

Detailed balance (with respect $\pi(\theta)$) is still preserved with α_{da} because

$$\begin{aligned} &\pi(\theta)\ q(\theta'|\theta)\ \alpha_{da}(\theta;\theta') \\ &\quad = \hat{\pi}_a(\theta)\ q(\theta'|\theta)\ \alpha_1 \ \times\ \tfrac{\pi(\theta)}{\hat{\pi}_a(\theta)}\ \alpha_2. \end{aligned}$$

Delayed acceptance MH (1)

(Christen and Fox, 2005). Suppose we have a computationally-cheap approximation to the posterior: $\hat{\pi}_a(\theta)$.

Define $\alpha_{da}(\theta;\theta') := \alpha_1(\theta;\theta')\ \alpha_2(\theta;\theta')$, where

$$\alpha_1 := 1 \wedge \frac{\hat{\pi}_a(\theta')q(\theta|\theta')}{\hat{\pi}_a(\theta)q(\theta'|\theta)} \quad \text{and} \quad \alpha_2 := 1 \wedge \frac{\pi(\theta')/\hat{\pi}_a(\theta')}{\pi(\theta)/\hat{\pi}_a(\theta)}.$$

Detailed balance (with respect $\pi(\theta)$) is still preserved with α_{da} because

$$\pi(\theta)\ q(\theta'|\theta)\ \alpha_{da}(\theta;\theta') = \hat{\pi}_a(\theta)\ q(\theta'|\theta)\ \alpha_1 \ \times\ \frac{\pi(\theta)}{\hat{\pi}_a(\theta)}\ \alpha_2.$$

But this algorithm mixes worse than the equivalent MH algorithm (Peskun, 1973; Tierney, 1998).

Delayed-acceptance [PsM]MH (2)

Using $\alpha_{da} = \alpha_1(\theta; \theta')\alpha_2(\theta; \theta')$ mixes worse but CPU time/iteration can be much reduced.

Delayed-acceptance [PsM]MH (2)

Using $\alpha_{da} = \alpha_1(\theta;\theta')\alpha_2(\theta;\theta')$ mixes worse but CPU time/iteration can be much reduced.

Accept $\Leftrightarrow$ accept at Stage 1 (w.p. α_1) and accept at Stage 2 (w.p. α_2).

Delayed-acceptance [PsM]MH (2)

Using $\alpha_{da} = \alpha_1(\theta;\theta')\alpha_2(\theta;\theta')$ mixes worse but CPU time/iteration can be much reduced.

Accept $\Leftrightarrow$ accept at Stage 1 (w.p. α_1) and accept at Stage 2 (w.p. α_2).

α_1 is quick to calculate.

Delayed-acceptance [PsM]MH (2)

Using $\alpha_{da} = \alpha_1(\theta; \theta')\alpha_2(\theta; \theta')$ mixes worse but CPU time/iteration can be much reduced.

Accept $\Leftrightarrow$ accept at Stage 1 (w.p. α_1) and accept at Stage 2 (w.p. α_2).

α_1 is quick to calculate.

There is no need to calculate α_2 if we reject at Stage One.

Delayed-acceptance [PsM]MH (2)

Using $\alpha_{da} = \alpha_1(\theta; \theta')\alpha_2(\theta; \theta')$ mixes worse but CPU time/iteration can be much reduced.

Accept $\Leftrightarrow$ accept at Stage 1 (w.p. α_1) and accept at Stage 2 (w.p. α_2).

α_1 is quick to calculate.

There is no need to calculate α_2 if we reject at Stage One.

If $\hat{\pi}_a$ is accurate then $\alpha_2 \approx 1$.

Delayed-acceptance [PsM]MH (2)

Using $\alpha_{da} = \alpha_1(\theta;\theta')\alpha_2(\theta;\theta')$ mixes worse but CPU time/iteration can be much reduced.

Accept $\Leftrightarrow$ accept at Stage 1 (w.p. α_1) and accept at Stage 2 (w.p. α_2).

α_1 is quick to calculate.

There is no need to calculate α_2 if we reject at Stage One.

If $\hat{\pi}_a$ is accurate then $\alpha_2 \approx 1$.

If $\hat{\pi}_a$ is also cheap then (RWM) can use large jump proposals [EXPLAIN].

Delayed-acceptance [PsM]MH (2)

Using $\alpha_{da} = \alpha_1(\theta;\theta')\alpha_2(\theta;\theta')$ mixes worse but CPU time/iteration can be much reduced.

Accept $\Leftrightarrow$ accept at Stage 1 (w.p. α_1) and accept at Stage 2 (w.p. α_2).

α_1 is quick to calculate.

There is no need to calculate α_2 if we reject at Stage One.

If $\hat{\pi}_a$ is accurate then $\alpha_2 \approx 1$.

If $\hat{\pi}_a$ is also cheap then (RWM) can use large jump proposals [EXPLAIN].

Delayed-acceptance PMMH:

$$\alpha_2 := 1 \wedge \frac{\hat{\pi}(\theta', u')/\hat{\pi}_a(\theta')}{\hat{\pi}(\theta, u)/\hat{\pi}_a(\theta)}.$$

Cheap and accurate approximation?

We: use an inverse-distance-weighted average of the π values from the k nearest neighbours.

Cheap and accurate approximation?

We: use an inverse-distance-weighted average of the π values from the k nearest neighbours.

But the cost is still $\mathcal{O}(n)/\text{iter}$.

k-nn and the binary tree

Imagine a table with n values.

$$\begin{array}{cccc|c} \theta_1^{(1)} & \theta_2^{(1)} & \cdots & \theta_d^{(1)} & \pi(\theta^{(1)}) \\ \theta_1^{(2)} & \theta_2^{(2)} & \cdots & \theta_d^{(2)} & \pi(\theta^{(2)}) \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \theta_1^{(n)} & \theta_2^{(n)} & \cdots & \theta_d^{(n)} & \pi(\theta^{(n)}) \end{array}$$

Look-up of k nearest neighbours to some θ' is $O(n)$.

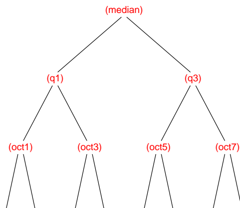
k-nn and the binary tree

Imagine a table with n values.

$$\begin{array}{cccc|c} \theta_1^{(1)} & \theta_2^{(1)} & \cdots & \theta_d^{(1)} & \pi(\theta^{(1)}) \\ \theta_1^{(2)} & \theta_2^{(2)} & \cdots & \theta_d^{(2)} & \pi(\theta^{(2)}) \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \theta_1^{(n)} & \theta_2^{(n)} & \cdots & \theta_d^{(n)} & \pi(\theta^{(n)}) \end{array}$$

Look-up of k nearest neighbours to some θ' is $O(n)$.

If $d = 1$ then could sort list or create a standard binary tree



for $\mathcal{O}(\log n)$ look up.

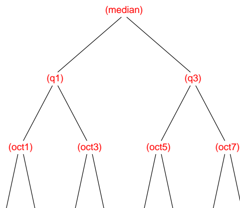
k-nn and the binary tree

Imagine a table with n values.

$$\begin{array}{cccc|c}
\theta_1^{(1)} & \theta_2^{(1)} & \cdots & \theta_d^{(1)} & \pi(\theta^{(1)}) \\
\theta_1^{(2)} & \theta_2^{(2)} & \cdots & \theta_d^{(2)} & \pi(\theta^{(2)}) \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
\theta_1^{(n)} & \theta_2^{(n)} & \cdots & \theta_d^{(n)} & \pi(\theta^{(n)})
\end{array}$$

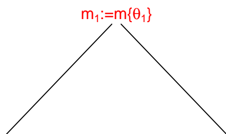
Look-up of k nearest neighbours to some θ' is $O(n)$.

If $d = 1$ then could sort list or create a standard binary tree



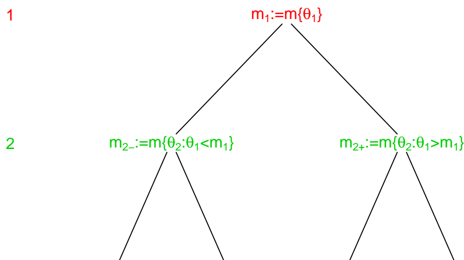
for $\mathcal{O}(\log n)$ look up. For $d > 1$ a solution is the KD-tree.

KD-tree (d=2)



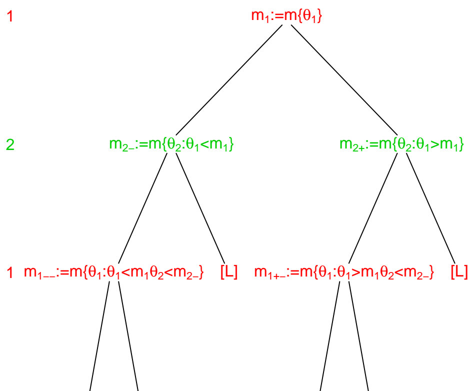
$m\{\mathcal{S}\} =$ branch splitting according to $\theta_{\text{d-split}}$ on median of $\mathcal{S}$.
$[L] =$ leaf node with a maximum of $2b - 1$ leaves.

KD-tree (d=2)



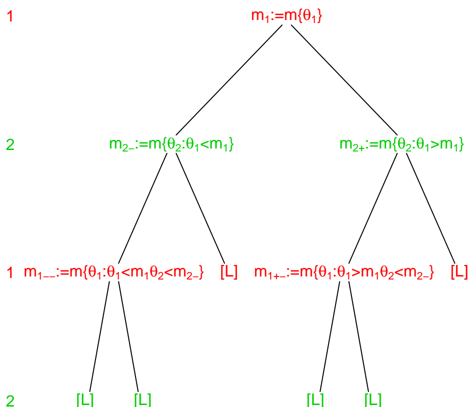
$m\{\mathcal{S}\} =$ branch splitting according to $\theta_{\text{d-split}}$ on median of $\mathcal{S}$.
$[L] =$ leaf node with a maximum of $2b - 1$ leaves.

KD-tree (d=2)



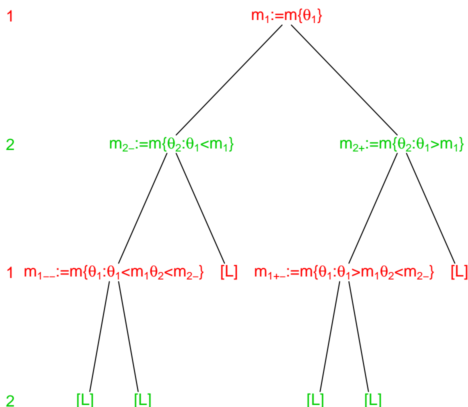
$m\{\mathcal{S}\} =$ branch splitting according to $\theta_{\text{d-split}}$ on median of $\mathcal{S}$.
$[L] =$ leaf node with a maximum of $2b - 1$ leaves.

KD-tree (d=2)



$m\{\mathcal{S}\} =$ branch splitting according to $\theta_{\text{d-split}}$ on median of $\mathcal{S}$.
$[L] =$ leaf node with a maximum of $2b - 1$ leaves.

KD-tree (d=2)



$m\{\mathcal{S}\} =$ branch splitting according to $\theta_{\text{d-split}}$ on median of $\mathcal{S}$.
$[L] =$ leaf node with a maximum of $2b - 1$ leaves.
Our KD-tree is useful if (roughly) $n/(3b/2) > 2^d$.

Adaptive k-nn using a KD-tree

Initial, training run of n_0 iterations. Build initial KD-tree.

Adaptive k-nn using a KD-tree

Initial, training run of n_0 iterations. Build initial KD-tree.

Main run: ‘every time’ $\pi(\theta')$ is evaluated, add $(\theta', \pi(\theta'))$ to the KD-tree.

Adaptive k-nn using a KD-tree

Initial, training run of n_0 iterations. Build initial KD-tree.

Main run: 'every time' $\pi(\theta')$ is evaluated, add $(\theta', \pi(\theta'))$ to the KD-tree.

Set-up is $O\left(n_0(\log n_0)^2\right)$; updating is $O(\log n)$ evaluation is $O(\log n)$ and accuracy $\uparrow$ as the MCMC progresses.

Adaptive k-nn using a KD-tree

Initial, training run of n_0 iterations. Build initial KD-tree.

Main run: 'every time' $\pi(\theta')$ is evaluated, add $(\theta', \pi(\theta'))$ to the KD-tree.

Set-up is $O\left(n_0(\log n_0)^2\right)$; updating is $O(\log n)$ evaluation is $O(\log n)$ and accuracy $\uparrow$ as the MCMC progresses.

Provided the tree is balanced. [Skip, for lack of time]

Refinements

Training dataset ⇒ better distance metric. Transform θ' to approximately isotropic before creating tree, or adding new node.

Refinements

Training dataset $\Rightarrow$ better distance metric. Transform θ' to approximately isotropic before creating tree, or adding new node.

Minimum distance ϵ. If $\exists\ \theta$ s.t. $||\theta' - \theta|| < \epsilon$ then
(i) MH: ignore new value.
(ii) PMMH: combine $\hat{\pi}(y|\theta', u')$ with $\hat{\pi}(y|\theta, u)$ (running average).

Adaptive Algorithm

Components

- A fixed [pseudo-marginal] kernel $P([\theta, u]; [\mathrm{d}\theta', \mathrm{d}u'])$.
- An adaptive [pseudo-marginal] DA kernel $P_\gamma([\theta, u]; [\mathrm{d}\theta', \mathrm{d}u'])$.

Adaptive Algorithm

Components

- A fixed [pseudo-marginal] kernel $P([\theta, u]; [d\theta', du'])$.
- An adaptive [pseudo-marginal] DA kernel $P_\gamma([\theta, u]; [d\theta', du'])$.
- Both P and P_γ generate $\hat{\pi}(\theta', u)$ in the same way.
- A fixed probability $\beta \in (0, 1]$.
- A set of probabilities: $p_n \to 0$.

Algorithm At the start of iteration n, the chain is at $[\theta, u]$ and the DA kernel would be P_γ.

Adaptive Algorithm

Components

- A fixed [pseudo-marginal] kernel $P([\theta, u]; [d\theta', du'])$.
- An adaptive [pseudo-marginal] DA kernel $P_\gamma([\theta, u]; [d\theta', du'])$.
- Both P and P_γ generate $\hat{\pi}(\theta', u)$ in the same way.
- A fixed probability $\beta \in (0, 1]$.
- A set of probabilities: $p_n \to 0$.

Algorithm At the start of iteration n, the chain is at $[\theta, u]$ and the DA kernel would be P_γ.

1. Sample $[\theta', u']$ from

$$\begin{cases} P & \text{w.p.} \quad \beta \\ P_\gamma & \text{w.p.} \quad 1 - \beta. \end{cases}$$

Adaptive Algorithm

Components

- A fixed [pseudo-marginal] kernel $P([\theta, u]; [d\theta', du'])$.
- An adaptive [pseudo-marginal] DA kernel $P_\gamma([\theta, u]; [d\theta', du'])$.
- Both P and P_γ generate $\hat{\pi}(\theta', u)$ in the same way.
- A fixed probability $\beta \in (0, 1]$.
- A set of probabilities: $p_n \to 0$.

Algorithm At the start of iteration n, the chain is at $[\theta, u]$ and the DA kernel would be P_γ.

1. Sample $[\theta', u']$ from
$$\begin{cases} P & \text{w.p.} \quad \beta \\ P_\gamma & \text{w.p.} \quad 1-\beta. \end{cases}$$
2. W.p. p_n 'choose a new γ': update the kernel by including all relevant information since the kernel was last updated.

Adaptive Algorithm

Components

- A fixed [pseudo-marginal] kernel $P([\theta, u]; [d\theta', du'])$.
- An adaptive [pseudo-marginal] DA kernel $P_\gamma([\theta, u]; [d\theta', du'])$.
- Both P and P_γ generate $\hat{\pi}(\theta', u)$ in the same way.
- A fixed probability $\beta \in (0, 1]$.
- A set of probabilities: $p_n \to 0$.

Algorithm At the start of iteration n, the chain is at $[\theta, u]$ and the DA kernel would be P_γ.

1. Sample $[\theta', u']$ from
$$\begin{cases} P & \text{w.p.} \quad \beta \\ P_\gamma & \text{w.p.} \quad 1 - \beta. \end{cases}$$
2. W.p. p_n 'choose a new γ': update the kernel by including all relevant information since the kernel was last updated.

Set: $p_n = 1/(1 + ci_n)$, where $i_n = \#$ expensive evaluations so far.

Ergodicity

Assumptions on the fixed kernel.

1. Minorisation: there is a density $\nu(\theta)$ and $\delta > 0$ such that $q(\theta'|\theta)\alpha(\theta;\theta') > \delta\nu(\theta')$, where α is the acceptance rate from the idealised version of the fixed MH algorithm.

Ergodicity

Assumptions on the fixed kernel.

1. Minorisation: there is a density $\nu(\theta)$ and $\delta > 0$ such that $q(\theta'|\theta)\alpha(\theta;\theta') > \delta\nu(\theta')$, where α is the acceptance rate from the idealised version of the fixed MH algorithm.
2. Bounded weights: the support of $W := \hat{\pi}(\theta, U)/\pi(\theta)$ is uniformly (in θ) bounded above by some $\overline{w} < \infty$.

Ergodicity

Assumptions on the fixed kernel.

1. Minorisation: there is a density $\nu(\theta)$ and $\delta > 0$ such that $q(\theta'|\theta)\alpha(\theta;\theta') > \delta\nu(\theta')$, where α is the acceptance rate from the idealised version of the fixed MH algorithm.
2. Bounded weights: the support of $W := \hat{\pi}(\theta, U)/\pi(\theta)$ is uniformly (in θ) bounded above by some $\overline{w} < \infty$.

Theorem Subject to Assumptions 1 and 2, the adaptive pseudo-marginal algorithm is ergodic.

Ergodicity

Assumptions on the fixed kernel.

1. Minorisation: there is a density $\nu(\theta)$ and $\delta > 0$ such that $q(\theta'|\theta)\alpha(\theta;\theta') > \delta\nu(\theta')$, where α is the acceptance rate from the idealised version of the fixed MH algorithm.
2. Bounded weights: the support of $W := \hat{\pi}(\theta, U)/\pi(\theta)$ is uniformly (in θ) bounded above by some $\overline{w} < \infty$.

Theorem Subject to Assumptions 1 and 2, the adaptive pseudo-marginal algorithm is ergodic.

NB. For DAMH, as opposed to DAPMMH, only the minorisation assumption is required.

Choice of β

DARWM is more efficient when $\lambda > \hat{\lambda}_{RWM}$.

Choice of β

DARWM is more efficient when $\lambda > \hat{\lambda}_{RWM}$.
The lower α_1 does not matter.

Choice of β

DARWM is more efficient when $\lambda > \hat{\lambda}_{RWM}$.
The lower α_1 does not matter.

But low $\alpha_1 \Rightarrow$ fewer expensive evaluations of π [or of $\hat{\pi}$].

Choice of β

DARWM is more efficient when $\lambda > \hat{\lambda}_{RWM}$.
The lower α_1 does not matter.

But low $\alpha_1 \Rightarrow$ fewer expensive evaluations of π [or of $\hat{\pi}$].

$$P(\text{expensive}) = \beta + (1 - \beta)\alpha_1.$$

Choice of β

DARWM is more efficient when $\lambda > \hat{\lambda}_{RWM}$.
The lower α_1 does not matter.

But low $\alpha_1 \Rightarrow$ fewer expensive evaluations of π [or of $\hat{\pi}$].

$$P(\text{expensive}) = \beta + (1 - \beta)\alpha_1.$$

If $\alpha_1 << 1$ most of the expensive evaluations come from the fixed kernel and much of the benefit from the adaptive kernel will be lost.

Choice of β

DARWM is more efficient when $\lambda > \hat{\lambda}_{RWM}$.
The lower α_1 does not matter.

But low $\alpha_1 \Rightarrow$ fewer expensive evaluations of π [or of $\hat{\pi}$].

$$P(\text{expensive}) = \beta + (1-\beta)\alpha_1.$$

If $\alpha_1 << 1$ most of the expensive evaluations come from the fixed kernel and much of the benefit from the adaptive kernel will be lost.

Fixing $\beta \propto \alpha_1$ (obtained from the training run) avoids this,

Choice of β

DARWM is more efficient when $\lambda > \hat{\lambda}_{RWM}$.
The lower α_1 does not matter.

But low $\alpha_1 \Rightarrow$ fewer expensive evaluations of π [or of $\hat{\pi}$].

$$P(\text{expensive}) = \beta + (1-\beta)\alpha_1.$$

If $\alpha_1 << 1$ most of the expensive evaluations come from the fixed kernel and much of the benefit from the adaptive kernel will be lost.

Fixing $\beta \propto \alpha_1$ (obtained from the training run) avoids this, but the guaranteed worst-case TVD from π after n iterations gets larger.

Choice of β

DARWM is more efficient when $\lambda > \hat{\lambda}_{RWM}$.
The lower α_1 does not matter.

But low $\alpha_1 \Rightarrow$ fewer expensive evaluations of π [or of $\hat{\pi}$].

$$P(\text{expensive}) = \beta + (1-\beta)\alpha_1.$$

If $\alpha_1 << 1$ most of the expensive evaluations come from the fixed kernel and much of the benefit from the adaptive kernel will be lost.

Fixing $\beta \propto \alpha_1$ (obtained from the training run) avoids this, but the guaranteed worst-case TVD from π after n iterations gets larger.

Consider a fixed computational budget $\approx$ fixed number of expensive evaluations. This preserves the provable worst-case TVD from π.

Examples

Lotka-Volterra MJP daPMRWM with $d = 5$

LNA approximation to autoregulatary system daRWM with $d = 10$

Examples

Lotka-Volterra MJP daPMRWM with $d = 5$

LNA approximation to autoregulatary system daRWM with $d = 10$

RWM: $\theta' \sim N(\theta, \lambda^2 \hat{\Sigma})$ where $\hat{\Sigma}$, obtained from training run (also gives pre-map).

Examples

Lotka-Volterra MJP daPMRWM with $d = 5$

LNA approximation to autoregulatary system daRWM with $d = 10$

RWM: $\theta' \sim N(\theta, \lambda^2\hat{\Sigma})$ where $\hat{\Sigma}$, obtained from training run (also gives pre-map).

Scaling, λ [and number of particles m] chosen to be optimal for RWM.

Examples

Lotka-Volterra MJP daPMRWM with $d = 5$

LNA approximation to autoregulatary system daRWM with $d = 10$

RWM: $\theta' \sim N(\theta, \lambda^2 \hat{\Sigma})$ where $\hat{\Sigma}$, obtained from training run (also gives pre-map).

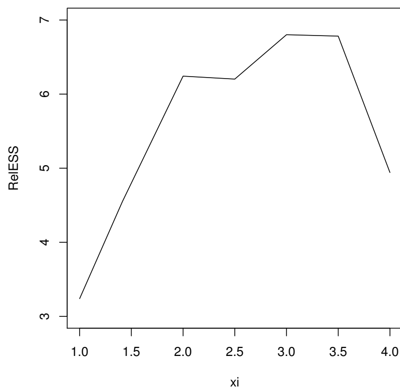
Scaling, λ [and number of particles m] chosen to be optimal for RWM.

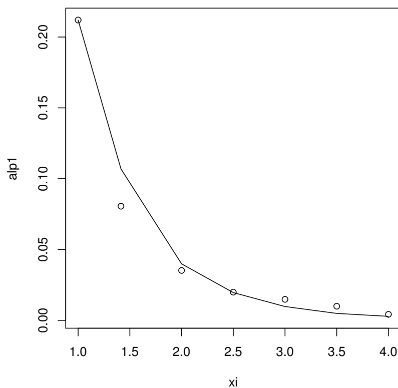
$n_0 = 10000$ (from training run), $b = 10, c = 0.001$.

$$\text{Efficency} \quad = \quad \frac{\min_{j=1\ldots d} \text{ESS}_j}{\text{CPU time}}$$

Results: LV

RelESS= efficiency of DA[PM]RWM / efficiency of optimal RWM.

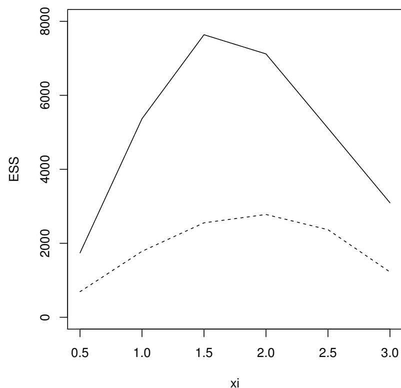


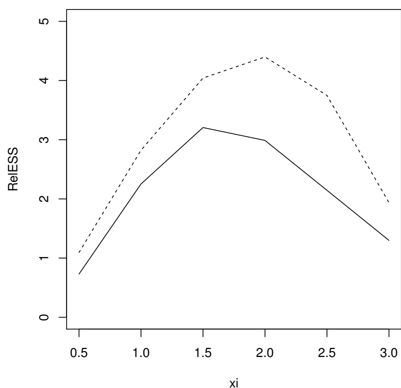


xi=1 corresponds to the DA using the scaling that is optimal for the standard RWM algorithm. i.e. DA scaling = xi$\times\hat{\lambda}_{RWM}$.

Results: Autoreg.

Solid=shorter dataset; dashed=longer dataset.





LV: further experiments

c	Tree Size	$\hat{\alpha}_1$	$\hat{\alpha}_2$	Rel. mESS
0.0001	41078	0.00772	0.339	7.28
0.001	40256	0.00915	0.276	6.80
0.01	43248	0.0121	0.204	4.67
∞	10000	0.0175	0.136	3.46

LV: further experiments

c	Tree Size	$\hat{\alpha}_1$	$\hat{\alpha}_2$	Rel. mESS
0.0001	41078	0.00772	0.339	7.28
0.001	40256	0.00915	0.276	6.80
0.01	43248	0.0121	0.204	4.67
∞	10000	0.0175	0.136	3.46

Using a list rather than the KD-tree reduced the efficiency by a factor of ≈ 2.

Summary

Store the expensive evaluations of π or $\hat{\pi}$ in a KD-tree.

Summary

Store the expensive evaluations of π or $\hat{\pi}$ in a KD-tree.

Use a delayed-acceptance [PsM]RWM algorithm.

Summary

Store the expensive evaluations of π or $\hat{\pi}$ in a KD-tree.

Use a delayed-acceptance [PsM]RWM algorithm.

Adaptive algorithm converges subject to conditions.

Summary

Store the expensive evaluations of π or $\hat{\pi}$ in a KD-tree.

Use a delayed-acceptance [PsM]RWM algorithm.

Adaptive algorithm converges subject to conditions.

Need $\beta \propto \alpha_1$ for adaptive portion to play a part.

Summary

Store the expensive evaluations of π or $\hat{\pi}$ in a KD-tree.

Use a delayed-acceptance [PsM]RWM algorithm.

Adaptive algorithm converges subject to conditions.

Need $\beta \propto \alpha_1$ for adaptive portion to play a part.

Improvement in efficiency by factor of between 4-7

Summary

Store the expensive evaluations of π or $\hat{\pi}$ in a KD-tree.

Use a delayed-acceptance [PsM]RWM algorithm.

Adaptive algorithm converges subject to conditions.

Need $\beta \propto \alpha_1$ for adaptive portion to play a part.

Improvement in efficiency by factor of between 4-7

Code easy-to-use, generic C code for the KD-tree is available.

Summary

Store the expensive evaluations of π or $\hat{\pi}$ in a KD-tree.

Use a delayed-acceptance [PsM]RWM algorithm.

Adaptive algorithm converges subject to conditions.

Need $\beta \propto \alpha_1$ for adaptive portion to play a part.

Improvement in efficiency by factor of between 4-7

Code easy-to-use, generic C code for the KD-tree is available.

Limitations: Need $n \gg 2^d$ for KD-tree to give worthwhile speedup and for adequate coverage.

Summary

Store the expensive evaluations of π or $\hat{\pi}$ in a KD-tree.

Use a delayed-acceptance [PsM]RWM algorithm.

Adaptive algorithm converges subject to conditions.

Need $\beta \propto \alpha_1$ for adaptive portion to play a part.

Improvement in efficiency by factor of between 4-7

Code easy-to-use, generic C code for the KD-tree is available.

Limitations: Need $n \gg 2^d$ for KD-tree to give worthwhile speedup and for adequate coverage.

Other: could use k nearest neighbours to estimate gradient and curvature;

Summary

Store the expensive evaluations of π or $\hat{\pi}$ in a KD-tree.

Use a delayed-acceptance [PsM]RWM algorithm.

Adaptive algorithm converges subject to conditions.

Need $\beta \propto \alpha_1$ for adaptive portion to play a part.

Improvement in efficiency by factor of between 4-7

Code easy-to-use, generic C code for the KD-tree is available.

Limitations: Need $n \gg 2^d$ for KD-tree to give worthwhile speedup and for adequate coverage.

Other: could use k nearest neighbours to estimate gradient and curvature; even to fit a local GP?

Summary

Store the expensive evaluations of π or $\hat{\pi}$ in a KD-tree.

Use a delayed-acceptance [PsM]RWM algorithm.

Adaptive algorithm converges subject to conditions.

Need $\beta \propto \alpha_1$ for adaptive portion to play a part.

Improvement in efficiency by factor of between 4-7

Code easy-to-use, generic C code for the KD-tree is available.

Limitations: Need $n \gg 2^d$ for KD-tree to give worthwhile speedup and for adequate coverage.

Other: could use k nearest neighbours to estimate gradient and curvature; even to fit a local GP?

Thank you!